

RL-TR-94-41
Final Technical Report
May 1994

AD-A280 275



1

PROTO CODE GENERATION TECHNIQUES

International Software Systems, Inc.

Dr. Ramon D. Acosta

DTIC
ELECTE
JUN 17 1994
S F D

94-18796



3998

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED.

*Copyright 1994 International Software Systems, Inc.
This material may be reproduced by or for the U.S. Government pursuant to the copyright license
under clause at DFARS 252.227-7013 (April 1988).*

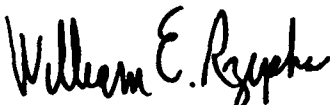
DTIC QUALITY INSPECTED 2


Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

94 6 16 071

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-41 has been reviewed and is approved for publication.

APPROVED: 
WILLIAM E. RZEPKA
Project Engineer

FOR THE COMMANDER: 
JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 1994		3. REPORT TYPE AND DATES COVERED Final Sep 91 - Jan 94
4. TITLE AND SUBTITLE PROTO CODE GENERATION TECHNIQUES			5. FUNDING NUMBERS C - F30602-91-C-0012 PE - 62702F PR - 5581 TA - 22 WU - 26	
6. AUTHOR(S) Dr. Ramon D. Acosta				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) International Software Systems, Inc. 9430 Research Blvd, Bldg 4, #250 Austin TX 78759-6543			8. PERFORMING ORGANIZATION REPORT NUMBER ISSI-C91A00013	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CB) 525 Brooks Road Griffiss AFB NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-94-41	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: William E. Rzepka/C3CB/(315) 330-2762				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Proto is an integrated system for specifying, analyzing, and validating software requirements for concurrent systems. Proto supports codesign and analysis of high-level software and hardware architectures early in the system life cycle. The system comprises a comprehensive tool set for editing hierarchical dataflow specification, object-based data modeling, component reuse, resource modeling, user-interface prototyping, software/hardware allocation, interactive simulation, and code generation. This report contains a summary of Proto's functional capabilities, languages, tools, and methodology.				
14. SUBJECT TERMS Software, rapid prototyping, object oriented, reuse, code generation, software/hardware allocation, interactive simulation, resource modeling			15. NUMBER OF PAGES 42	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

ABSTRACT.....	1
1 INTRODUCTION	1
2 FUNCTIONAL OVERVIEW	3
3 SOFTWARE AND HARDWARE PROTOTYPING LANGUAGES.....	6
3.1 SOFTWARE SPECIFICATION USING SSDL	6
3.2 HARDWARE SPECIFICATION	9
4 PROTO TOOLS	11
4.1 SYSTEM DESCRIPTION.....	11
4.1.1 User Interface Manager	14
4.1.2 Object Manager.....	14
4.2 TOOLS.....	15
4.2.1 Project Manager.....	15
4.2.2 Graph Editor	15
4.2.3 Behavior Editor.....	18
4.2.4 Type Editor	18
4.2.5 Variable Editor.....	18
4.2.6 Dynamic Loader	19
4.2.7 Rapid Interface Prototyping System.....	19
4.2.8 Reuse Library Manager	20
4.2.9 Export Facility	20
4.2.10 Import Facility	20
4.2.11 Architecture Editor	20
4.2.12 Mapping Editor	21
4.2.13 Interpreter.....	21
4.2.14 Ada Code Generator	23
5 METHODOLOGY OVERVIEW.....	23
6 EXAMPLE	27
7 CONCLUSIONS	30
8 REFERENCES.....	31

Codes

Dist	Avail and/or Special
A-i	

ABSTRACT

Proto is an integrated system for specifying, analyzing, and validating software requirements for concurrent systems. The system was developed under Air Force Rome Laboratory Contract No. F30602-91-C-0012, Proto Code Generation Techniques. Proto supports codesign and analysis of high-level software and hardware architectures early in the system life cycle. The system comprises a comprehensive tool set for editing hierarchical dataflow specifications, object-based data modeling, component reuse, resource modeling, user-interface prototyping, software/hardware allocation, interactive simulation, and code generation. This report contains a summary of Proto's functional capabilities, languages, tools, and methodology.

1 INTRODUCTION

Requirements are precise statements of need intended to convey a specific understanding about a desired result. Regardless of whether requirements specify needs for airline reservation systems, process control software, or automobiles, ideally they describe external, user-visible characteristics rather than internal system structure. Requirements also specify the constraints placed on what is needed. Performance, reliability, safety, cost, schedule and the reuse of existing designs are typical constraints. Requirements engineering is the activity of forming a model based on the requirements and then validating that the model accurately represents what is needed [Rzepka and Ohno 1985].

The importance of requirements engineering for software systems has been widely recognized in many application domains, including medical diagnosis, command and control, information systems, avionics, and emergency management. The inherent complexity of establishing an unambiguous, complete, and consistent set of requirements prior to system design and construction has led software engineering researchers and practitioners to propose and adopt rapid prototyping techniques and spiral development as part of the requirements engineering process.

Rapid prototyping techniques support specification and design analysis of critical software elements early in the system lifecycle [Boehm 1988; Gordon and Bieman 1991]. Development of evolutionary prototypes, with continuous user involvement and review, can provide significant improvements to the requirements specification process for complex software systems. The importance of improving this process cannot be overemphasized: doing so will reduce the risk, cost, and time associated with developing a software system and will lead to improved quality, reliability, reusability, and maintainability of the resultant system.

The greatest payoff in employing rapid prototyping techniques for requirements engineering is obtained by using environments integrated around a common underlying representation model that supports requirements elicitation and capture tightly coupled with high-level executable specification languages. Such an environment facilitates the electronic transfer of requirements into the prototyping context to initiate the design and development process. Progress in developing rapid prototyping environments, however, has been slowed by the lack of unifying models and technology capable of representing the complex data relationships associated with requirements. Moreover, few attempts have been made in extending rapid prototyping techniques for requirements specification and design of concurrent, parallel, and distributed systems.

The Requirements Engineering Environment (REE) provides an integrated toolset for rapid construction of executable prototypes to model and analyze software system requirements. REE contains interactive simulation-based facilities to allow systems analysts to rapidly represent,

build, and execute models of critical aspects of complex systems [Rzepka et al. 1993]. Functional, user interface, and performance models can be constructed easily, and at varying levels of abstraction or granularity, depending on the specific behavioral aspects of the system being exercised. REE is being developed as part of Rome Laboratory's research program in requirements engineering which has been in place since 1985 [Rzepka 1992].

The major components of REE include Proto, the Rapid Interface Prototyping System (RIP), and an interface routine package that integrates Proto and RIP. Proto is a rapid prototyping computer-aided software engineering (CASE) system that supports functional and performance prototyping of systems incorporating both sequential and parallel processing elements. RIP is a collection of tools that support building, executing, and analyzing user interface prototypes. Access to all of the RIP capabilities is accomplished through graphic, menu, and template driven interfaces, allowing requirements engineers who are not programmers to readily utilize the system.

This report concentrates on describing the functional capabilities, languages, tools, and methodology of Proto. It provides a complete summary of the system and serves a starting point for readers interested in understanding Proto and its underlying rapid prototyping technology.

In addition to this report, a full complement of documentation materials is available for gaining a greater understanding of using Proto and its underlying implementation:

- *Functional Description for Proto* [Acosta 1993a]
- *System/Subsystem Specification for Proto* [Acosta 1993b]
- *Program Specification for Proto* [ISSI 1994]
- *Proto User's Manual, Volumes 1 and 2* [ISSI 1993d]
- *Proto Methodology* [Acosta and Beal 1994]

Additional documentation associated with the project includes [Acosta and Liu 1993; ISSI 1993b, 1993c; Cechovic 1993; Box 1993].

The main source of information for those wishing to explore the user interface prototyping facilities of RIP are referred to:

- *Software User's Manual for the Rapid Interface Prototyping (RIP) System of the Requirements Engineering Environment* [ISSI 1993a]

Titles of recent papers that provide more in-depth coverage of specific functional and implementation aspects of Proto, RIP, and REE which may be of interest to readers include:

- *Parallel Proto - A Prototyping Tool for Analyzing and Validating Sequential and Parallel Processing Software Requirements* [Burns 1991]
- *A Requirements Engineering Testbed: Concept and Status* [Rzepka 1992]
- *Specification Prototyping of Concurrent Ada Programs in Proto* [Acosta 1992]
- *Requirements Engineering Technologies at Rome Laboratory* [Rzepka et al. 1993]
- *Use of Simulation Techniques in a Prototyping Environment for Requirements Engineering* [Sidoran and Acosta 1993]

- *A Case Study of Applying Rapid Prototyping Techniques in the Requirements Engineering Environment* [Acosta et al. 1994]

Subsequent sections of this report are structured as follows. In Section 2, an overview of the Proto functionality is provided. Section 3 describes the software and hardware prototype specification languages. Section 4 contains a high-level description of the Proto system and its tools. An overview of the Proto prototyping methodology is presented in Section 5. Section 6 contains a simple example to suggest how Proto facilities are employed for rapid prototyping of requirements. Finally, Section 7 contains summarizing conclusions and directions for future work.

2 FUNCTIONAL OVERVIEW

The goal of the Proto Code Generation Technique contract is to develop *Proto*, a rapid prototyping computer-aided software engineering (CASE) system that supports specification and design of systems incorporating both sequential and parallel processing elements [RADC 1990]. The system supports codesign and analysis of high-level software and hardware architectures early in the system life cycle. Proto comprises a comprehensive tool set for window-based graphical editing of hierarchical dataflow graph specifications, object-based data modeling, component reuse, graphical hardware resource modeling, user-interface prototyping, software/hardware allocation, interpreted execution using simulation, functional animation, interactive debugging, and code generation. The system is built on top of a substrate for managing user interfaces and database objects to provide consistent views of design objects across system tools.

Proto builds upon previous work sponsored by Rome Laboratory, including:

- **Proto.** Feasibility prototype of a rapid functional prototyping environment [Hartman et al. 1988].¹
- **Proto+.** Prototyping tool for specification and design of software systems based on dataflow concepts [ISSI 1992].
- **PProto (Parallel Proto).** Upward-compatible extension of Proto+ with capabilities for hardware architecture modeling and software/hardware allocation [Acosta 1991].

Proto provides a design environment for *systems analysts* through the use of a high-level prototyping language called the System Specification and Design Language (SSDL). In addition, it incorporates significant domain-specific reuse and code-generation capabilities necessary to support *domain users*. Other areas of emphasis in Proto include modeling tools for parallel and distributed processing, user interface prototyping, and methodological foundations. An important goal of the project is to validate the effectiveness of the Proto methodology and tools by constructing two domain-specific reuse libraries and a demonstration of a representative Air Force application.

SSDL is a high-level prototyping language based on visual dataflow abstractions. In SSDL, a clear distinction is made between the following portions of a design:

- **Dataflow Specification.** Graphical model of a prototype.

1. Note that the term Proto, which was used as the name of the original VHLL prototyping system, is being used as the name of the system described in this document (which is an enhancement of Proto/Proto+/PProto). When this project started, DProto was used to refer to the enhancements system.

- **Behavior Specification.** Data transformations effected by leaf nodes of a dataflow graph.
- **Data Specification.** Dynamic object-based data definitions.

SSDL contains many features that are useful in building prototype models of concurrent, parallel, and distributed systems, including several message-passing protocols and shared data stores that facilitate describing systems containing concurrent cooperating processes. Data modeling in SSDL is accomplished with a highly flexible object-based design paradigm. Dynamic modification of object features, such as relations to other objects and inheritance, is simple and valuable for rapid prototyping. Behaviors can access user-defined C functions, which allows SSDL to be used for defining prototypes as compositions of previously developed components.

Functional capabilities of Proto are summarized below:

Software Prototyping. Software systems are specified and designed using an enhanced version of the SSDL prototyping language. SSDL is a hierarchical dataflow language that provides constructs for describing visual dataflows, object-based data models, and textual behaviors. SSDL has concurrent execution semantics, which support simulation-based interpretation of specifications and code generation. Software modeling primitives include process nodes, connections, ports, and stores. These components can be configured to model a wide variety of concurrent systems, including those whose communication and synchronization is based on message passing or on shared memory. Explicit user-specified control connection mechanisms to enable node execution are also available. Data types include integers, reals, strings, enumerations, classes (with attributes, methods, and inheritance), and arrays. SSDL designs are organized into independent *projects* under user control.

A sophisticated multiwindow dataflow graph editor supports rapid construction of SSDL prototypes. This editor provides integrated access to facilities for reuse, software/hardware allocation, editing and parsing of behaviors, and editing of variables and types. The editors for variables and types employ graphical and menu-based displays for supporting development of complex data models. All of the SSDL editing tools are integrated via an underlying structure of semantic database objects that is accessed by other tools of the system.

Component Reuse. Management and browsing of libraries containing domain-specific reusable components are integrated with the system editors. The primary reusable components consist of leaf and hierarchical library process nodes, and user-specified data types. Reusable data types include classes (with their methods), enumerations, and arrays. Capabilities that assist in component reuse include explicit control connections, cut/copy/paste of class method nodes and library process nodes, and keyword search. Systems analysts can access facilities for developing reusable SSDL prototype components and adding the components to domain-specific libraries. Libraries can also be combined by invoking a merge operation. Domain users are supported by capabilities to modify and integrate reusable components into domain-specific prototypes.

Another capability that supports reuse is a dynamic loading tool for linking externally developed C functions into a prototype under construction. These functions can be called from SSDL behaviors. Proto also contains export and import tools that use a file-based ASCII format for representing SSDL specification prototypes. This format is a convenient mechanism for transferring designs between databases, sending designs using electronic mail, cleaning up databases, performing regression testing, and providing an interface between Proto and other tools.

User Interface Prototyping. The system incorporates the REED RIP tool [ISSI 1993] to construct graphical input and output interfaces for SSDL prototypes. The user is provided with an functional application programmer interface (API), termed DRI, to manipulate RIP objects by referencing and modifying their state attributes from SSDL behaviors. In this manner, generic and application-specific user-interface prototypes can be developed in conjunction with functional prototypes. RIP supports not only interface graphics but also forms-based devices, such as buttons and text fields.

Architecture Modeling. Proto provides a generic mechanism for specifying several kinds of MIMD architectures. These include shared-memory multiprocessors, distributed-memory multiprocessors, and hybrid machines consisting of shared-memory and distributed-memory components. Hardware modeling primitives include processors, memories, and buses. User-specified parameters are supplied for selection of hardware resource characteristics, including machine topology, processor execution speed, memory-access time, and bus delays. Hardware component timings are specified in terms of simulation time units. An algorithm for automatically computing static message routing is also provided.

Architectural features are described with a graphical architecture definition language. Editing of this visual language is accomplished by a graphical editor. The editor also supports automatic layout of parameterized mesh architectures containing an arbitrary number of rows and columns of processors interconnected by buses. Architecture specifications are stored as database entries for subsequent use with mapping and simulation tools.

An editor for allocating software (logical) components to hardware (physical) components is available. This editing facility is integrated with the dataflow graph editor. A mapping consists of pairs of software and hardware components taken from their respective definitions. This is accomplished by associating graphical objects that denote software activities with the graph that captures the actual hardware. Thus, a mapping is an embedding of the graph of logical connections into the graph of physical connections. More than one mapping per software specification is possible. Automatic heuristic and random allocation algorithms are also provided.

Simulation-Based Prototype Execution. Subsystems to support interactive execution of prototypes include an SSDL interpreter, a scheduler, and an architecture modeler. These functions are controlled by a simulation kernel that implements a simulation cycle using a time-based event queue. The interpreter directly executes SSDL behaviors. Using a time-based algorithm, the scheduler arbitrates among process nodes that are competing to execute on a single processor. Software design simulation is constrained by an architecture resource model according to the software/hardware mapping that is specified. The architecture modeler manages resource simulation, including message routing, serialization of memory and bus requests, and display of utilization statistics.

Interactive debugging and performance evaluation features provided in Proto include dataflow graph animation, node breakpoints, single-stepping of behaviors, dynamic data display instruments, dynamic resource utilization instruments, and object browsing facilities. In addition, functions for debugging concurrent specifications are provided. One of these event-based tools implements automatic detection of deadlocks. A generic conditional breakpoint facility provides support for detection of exceptional concurrent event sequences, such as data corruption and data unavailability.

Code Generation. In support of specification and design for both systems analysts and domain users, Proto incorporates an automatic code generation tool for generating Ada implementations from SSDL prototypes. Multitasked Ada code that can be easily integrated with other systems is generated for all components of an SSDL specification prototype. Communication and synchronization between process nodes are accomplished with Ada's rendezvous mechanism, thus providing a model appropriate for distributed applications. The generated code employs the same scheduling algorithm as the Proto simulator, which results in code that is semantically equivalent to the original SSDL, except for timing dependencies and inherent nondeterminism.

User Interface Management. Interfaces to the editing, simulation, and code generation tools are managed by a common graphical interface substrate that implements the OPEN LOOK user interface standard. The user interface manager guarantees a clear and consistent interface across tools.

Object Management. Management of persistent design objects is implemented by a commercial object-oriented database commonly accessible to all system tools. Examples of objects stored in the database include SSDL visual representations, semantic structures, resource architectures, and reuse libraries.

3 SOFTWARE AND HARDWARE PROTOTYPING LANGUAGES

Proto employs SSDL as its notation for specifying the prototype of a software component. A specialized hardware architecture definition notation is used to define resource prototypes. This section describes these two representations.

3.1 SOFTWARE SPECIFICATION USING SSDL

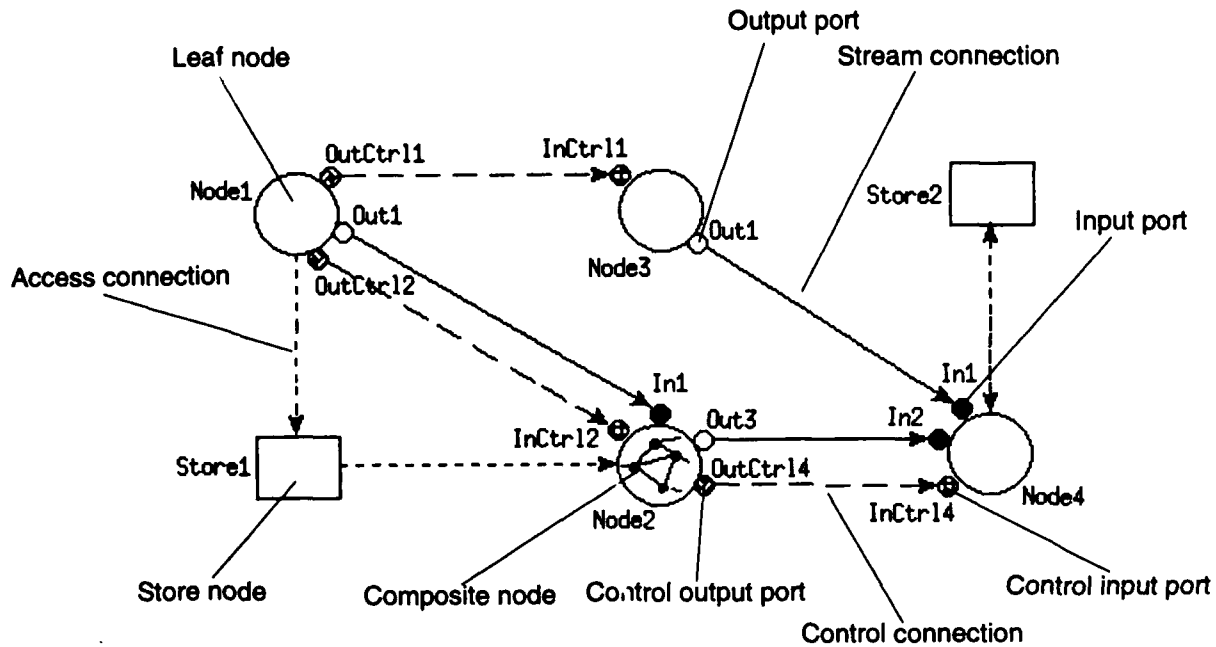
Software system prototype specifications in SSDL are modeled as hierarchical dataflow graphs that contain the following primitives (Figure 1):

- **Process Nodes.** Sequential or concurrent processes, functions, operators
- **Stores.** Shared memory to store state information
- **Connections.** Communication channels between process nodes and stores
- **Ports.** Interfaces between process nodes and connections

Each process node in the hierarchy may be structurally specified by a decomposition, or refinement, which is itself a dataflow graph. This nested graph receives all its inputs from and sends all its outputs to the ports on its enclosing node. The *behavior* of each leaf node of a graph defines the functional transformation that takes place inside a process node.

A dataflow description corresponds to a functional specification, where dataflow process nodes denote system functions. Data is transient and passes from node to node along connections in the form of messages. It is possible, however, to save data across process node executions using stores. Thus, there are two kinds of data in SSDL: *activation data* that fires the output-triggering data-transforming computation inside a process node, and *reference data*, resident in stores, that is read and written by a process node and is preserved across node executions. This basic computation model of concurrent cooperating processes thus supports both shared-variable and message-passing communication styles.

Parent Graph



Refinement of Node2

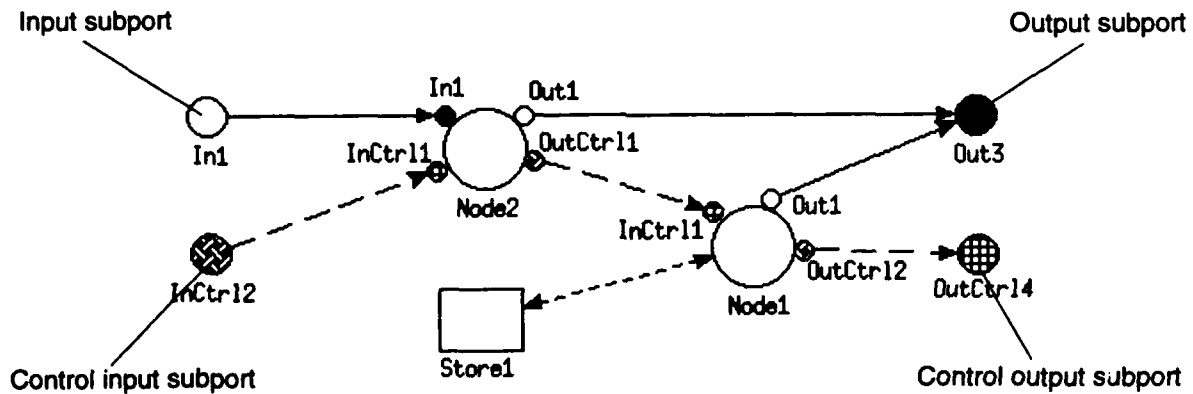


Figure 1. Graphical Symbols Used in SSDL

Concurrency is introduced into an SSDL model by mapping process nodes onto different *processors* defined in the hardware architecture model. A process node, which is always mapped to some processor, can execute in parallel with nodes mapped onto other processors as long as (1) its activation data is available, (2) the processor it is mapped to is idle, and (3) it has higher precedence than other activated process nodes that are mapped onto its same processor. Precedence is determined by a *scheduling algorithm* that arbitrates among multiple process nodes competing to execute on an single processor. This algorithm considers node activation times, execution counts, and node priorities in making the scheduling decision.

Data modeling uses the object-based concepts of *class* and *instance*. A class is a type definition of a set of objects that defines attributes, or slots, that describe the features of members of the set. A class can have subclasses that inherit attributes from their parent class and can have additional attributes not present in their parent class. An instance of a class is a specific application object that belongs to the set of objects defined by the class and contains a unique version of its inherited attributes. In addition to classes, other types supported by SSDL include integers, reals, strings, arrays, and enumerations. An *unspecified* type is also provided which, although requiring dynamic type checking during execution, allows delaying type constraint decisions during the prototype construction process.

SSDL supports three kinds of *variables*: ports, stores, and locals. By default, variables are of the unspecified type, i.e., they can assume values of any type. Stores are variables that preserve state across process node executions. Stores are also marked with a *lock* attribute that can be used to restrict concurrent access in such a manner as to prevent inadvertent shared data corruption. Node ports are denoted as being either input ports or output ports. Input ports have attached first-in-first-out (FIFO) queues that store incoming data. Ports are visible to their containing process node and its decomposition, if any. Visibility of locals is strictly limited to their containing process node.

Asynchronous communication between process nodes is available using *stream* connections, which provide unbounded buffer channels. Synchronous communication is modeled with *synchronized* connections between process nodes, which provide automatic generation of acknowledgments when node behaviors that process the incoming "synchronized" data messages complete execution. *Sampled* communication connections, which implement a sample-and-hold communication protocol, are also available. Read and write access from process nodes to stores is achieved over explicit *access* connections.

Sequential or concurrent process node behaviors in SSDL are described with a simple structured programming notation whose statements are interpreted by the Proto simulator. Behaviors consist of one or more *behavior rules*, which contain a *trigger*, or guard, and an *action*. Thus, the general structure of a node behavior is the following:

initial

-- Port queue and store initialization.

end initial

behavior

<trigger1> :- <action1> !

<trigger2> :- <action2> !

...

```

    <triggerN> :- <actionN> !
end behavior

```

If a trigger is empty, the behavior rule is said to be an *independent* behavior, otherwise, the behavior rule is a *dependent* behavior. Independent behavior rules do not depend on input data and are always eligible to execute. Conditional trigger expressions for dependent behaviors specify how data messages are accepted, or received, by a process node (i.e., **accept** operations) and the input data conditions under which a process node is activated. Even though the triggers of multiple behavior rules within a process node may be true simultaneously, at most one behavior rule within a process node can be executing at any given time. In this manner, use of the behavior rule construct allows specification of nondeterministic selection of alternative actions, which subsequently influences scheduling decisions during simulation and generated code execution.

High-level SSDL constructs for specifying behavior rule actions include iteration, conditional statements, assignment, message-passing primitives (i.e., **send** operations), and variable accesses. Behavior constructs are also available to initialize port queues and stores, and provide access to external procedures written in C. A functional interface to a comprehensive math library is also available.

A **read** operation for a variable is carried out whenever the variable is referenced in evaluating an expression (e.g., the expression on right-hand-side of an assignment statement or a conditional expression in an iteration statement). Variable **write** operations occur whenever the variable appears on the left-hand-side of an assignment statement.

3.2 HARDWARE SPECIFICATION

Proto provides a generic mechanism for specifying parallel and distributed MIMD architectures. These include shared-memory multiprocessors, distributed-memory multiprocessors, and hybrid machines consisting of shared-memory and distributed-memory components. Hardware resource modeling primitives include (Figure 2):

- **Processors.** Execute process node behaviors
- **Memories.** Contain store values
- **Buses.** Transfer messages for node communication and store accesses

Users are free to specify any machine topology consisting of these components. The only topological restrictions are that buses must be used to connect processors to other processors or to memories; i.e., processors and memories cannot be directly connected to each other. Proto also provides a mechanism for automatically generating mesh architectures consisting of rows and columns of processors interconnected by buses.

User-specified parameters for selection of hardware resource characteristics include processor execution speed, read and write memory-access times, and message delays. All of these parameters are defined in terms of simulation time units. Specification of processor execution speed involves designating delays for the following behavior operations:

- **add/subtract** – addition and subtraction (+, -)
- **accept** – accepting from stream, sampled, and synchronized connections

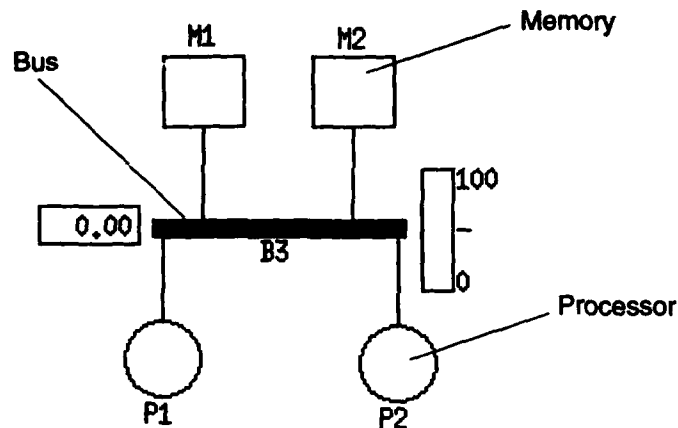


Figure 2. Graphical Symbols Used in Architecture Definitions

- **assign** – writing of port, store, and local variables (**:=**)
- **c_call** – calls to predefined and external user-defined C functions
- **divide** – division (**/**)
- **jump** – **if**, **loop**, and **exit** statements
- **index** – array indexing (**[]**)
- **logical_op** – logical operators (**not**, **and**, **or**)
- **multiply** – multiplication (*****)
- **new** – creation of new class instances (**new** and **copy** operators)
- **read** – reading of port, store, and local variables, and constants
- **relational_op** – relational operators (**>**, **>=**, **<**, **<=**, **=**, **!=**)
- **attribute** – access to class instance attributes (**.**)
- **send** – sending to stream, sampled, and synchronized connections
- **stop** – **stop** statement

Static minimum delay routing between primitives is automatically computed by the system using Floyd's shortest path algorithm [Aho et al. 1983]. During simulation, these routings are used to process message-passing (**send**) and data-store access (**read/write**) requests as high-level constructs. Consequently, a software design does not have to be changed when it is ported from one architecture to another, or when different software/hardware mappings are used for the same architecture. For example, a software specification that assumes a logical shared-memory model

can be mapped onto a shared-memory multiprocessor or a distributed-memory multiprocessor without changing any code. In the latter case, the physical model is one of distributed shared memory.

The system includes a simple mechanism for mapping software (logical) components to hardware (physical) components. A mapping consists of pairs of software components (processes and stores) and hardware components (processors and memories) taken from their respective definitions. Such a mapping results in an embedding of the dataflow graph of logical connections into the architecture graph of physical connections. A simple illustration of the flexibility afforded by having distinct software and hardware models with explicit allocation appears in Figure 3.

4 PROTO TOOLS

This section describes Proto in terms of its user tools and high-level architecture. These tools are used to build software and hardware prototype models, perform simulations, and generate code.

4.1 SYSTEM DESCRIPTION

Figure 4 depicts the high-level implementation architecture of Proto. The system is primarily implemented in C++ and built on top of the Unix operating system. There are also a number of implemented Ada modules for support of the code generation tool. Database management services are provided by the Object Manager (OM), which is based on a commercial object-oriented database. The User Interface Manager (UIM) supplies graphical and textual interface services to most system tools. This manager is based on capabilities of the X Window System server, which provides low-level, device-independent network graphics services.

Proto contains 14 tools, which are categorized into six functional areas and briefly described below. More complete explanations of the tool functions appear in subsequent sections.

Software Prototyping

- **Project Manager.** Creation, renaming, and deletion of SSDL prototypes
- **Graph Editor.** Multiwindow graphical editing of SSDL dataflow graphs
- **Behavior Editor.** Textual editing of SSDL process behaviors
- **Type Editor.** Multiwindow menu-based and graphical editing of object-based types
- **Variable Editor.** Multiwindow menu-based editing of scoped variable specifications

Component Reuse

- **Reuse Library Manager.** Management and access to SSDL component libraries
- **Dynamic Loader.** Loading and linking of external C routines
- **Export Facility.** Generation of an ASCII file representation for a project
- **Import Facility.** Recreation of a project from an ASCII file representation

User Interface Prototyping

- **Rapid Interface Prototyping System.** REED tool that provides graphical definition of application user interface prototypes

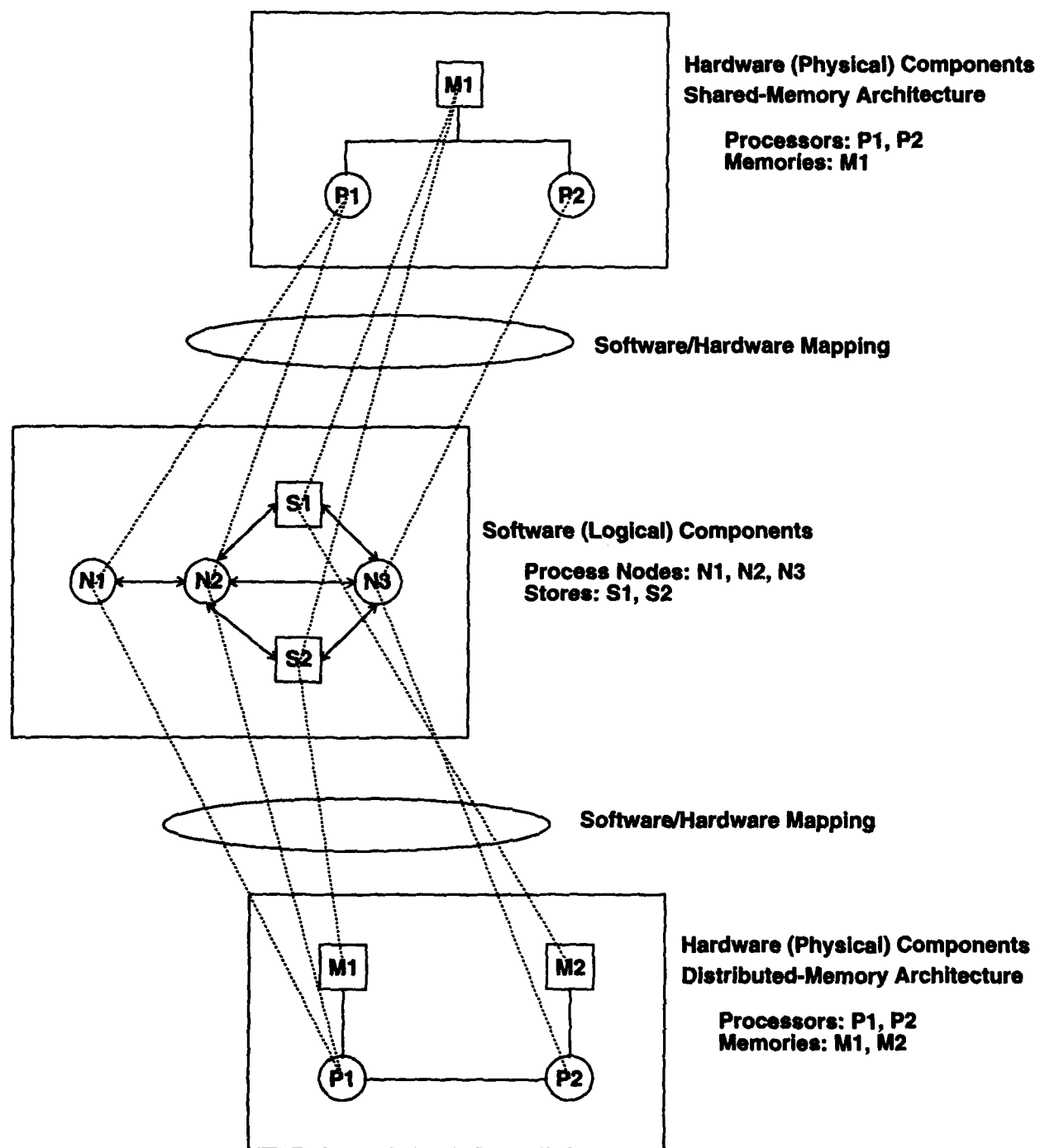


Figure 3. Software/Hardware Allocation Example

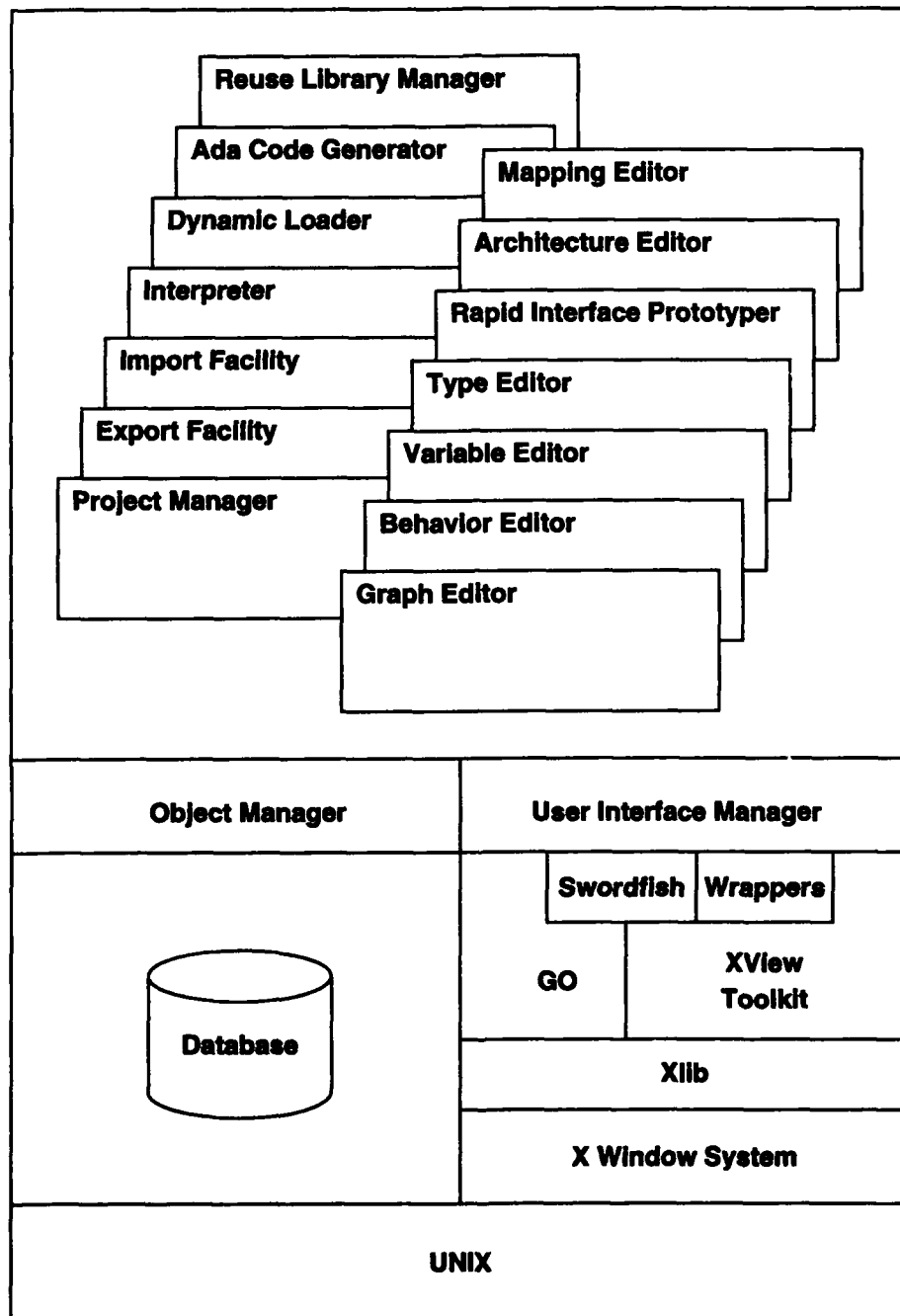


Figure 4. Proto Implementation Architecture

Architecture Modeling

- **Architecture Editor.** Graphical editing of multiple instruction, multiple data stream (MIMD) hardware component topologies
- **Mapping Editor.** Software/hardware component mappings

Simulation-Based Prototype Execution

- **Interpreter.** Functional execution and performance evaluation using simulation, animation, and debugging mechanisms

Code Generation

- **Ada Code Generator.** Generation of multitasked Ada from SSDL specification prototypes

4.1.1 User Interface Manager

The UIM collects graphical and textual user inputs and distributes them to appropriate tools. The UIM also manages the multiwindow display screen. In this manner, Proto keeps a clean separation between user-interface functions and application functions that operate on application data. Consequently, the system shields tool applications from modifications and enhancements to the user interface.

The UIM provides Proto tools direct access to

- **GO (Graphic Objects).** C++-based graphic object manipulation package that references graphics rendering capabilities obtained from Xlib. GO gives application developers access to higher-level graphical primitives and editing operations than those available with Xlib and the X Window System standard.
- **XView Toolkit.** Sun Microsystems presentation-layer library widgets (user interface objects) for window management that facilitate tool integration based upon the OPEN LOOK protocol standard.
- **Wrappers.** C++-based object interface package that references the widget manipulation capabilities obtained from the XView Toolkit. Wrappers gives application developers a higher-level interface to OPEN LOOK style widgets than is available from the XView Toolkit.
- **Swordfish.** Graph editor builder capability that supports construction of customizable graph editors from descriptions containing information about abstract-graph connectivity, icon-based renderings, and C++ semantic classes.

4.1.2 Object Manager

Proto uses an object-oriented database, or object base, to manage persistence and sharing of specification artifacts, including SSDL prototypes, architecture descriptions, and reuse libraries. The OM provides an application-level interface for controlled access to the database. Proto supports simultaneous access to multiple databases by multiple users in order to acquire reusable component libraries. Because of database limitations, however, opening of a database is limited to a single user at any given time. The OM is based on the ONTOS Object Database, a commercial

object-oriented database for C++ class instances. A save capability for explicitly storing user data in the database is available from most Proto editing tools.

4.2 TOOLS

Figure 5 shows Proto's logical block diagram. Six of the Proto tools are identified as *executive tools* in Figure 6, which illustrates primary access paths between tools. Executive tools can be accessed directly from Proto's top-level interface menu; that is, any executive tool is directly accessible from any other executive tool. User access to other tools is indirect based on context. For example, the Type Editor can be invoked directly from the Graph, Behavior, or Variable Editor or the Reuse Library Manager, but access from the Project Manager is indirect through the Graph Editor or the Reuse Library Manager.

The only direct access to RIP from Proto is through the Behavior Editor. This access is accomplished by embedding calls to an API functional interface inside SSDL process behaviors. RIP is implemented as a stand-alone executable module and, thus, is typically invoked independently from Proto as a separate Unix process.

This section provides more detailed descriptions of the Proto tools.

4.2.1 Project Manager

Proto's top-level interface for managing SSDL prototypes, also referred to as projects, is the Project Manager tool. The Project Manager allows creating, deleting, and renaming of projects. It also provides access to the Export and Import Facilities for saving and retrieving projects using ASCII file representations. The Reuse Library Manager is also invoked from the Project Manager. A simple utility that prints Proto version information, including support software versions, is also available in the Project Manager.

Whenever a project is created, a corresponding project library is also created. The project library, which can be manipulated as a conventional library from the Reuse Library Manager, contains all of the library components (data types and library process nodes) that can be accessed from the prototype graph of a project. A project thus consists of a prototype graph together with a project library.

4.2.2 Graph Editor

Dataflow graphs in SSDL are created and modified with a multiwindow Graph Editor. This is an object-oriented editor in that a user first selects an object (i.e., node, store, connection, port) and then selects the action to be performed on that object. Graphical object operations include inserting, moving, deleting, labeling, connecting, and aligning objects. The editor supports composition of nodes into hierarchical dataflow refinements. Other graphical facilities available in the editor include user-specified instrumentation, panning, zooming, and scrolling.

The Graph Editor provides integrated access to other tools, including the Behavior Editor, Type Editor, Variable Editor, Reuse Library Manager, and Dynamic Loader. Graphical cut-and-paste and copy-and-paste operations are provided for reuse purposes. In addition, the Graph Editor windows are used during simulation for prototype animation, instrumentation, and debugging.

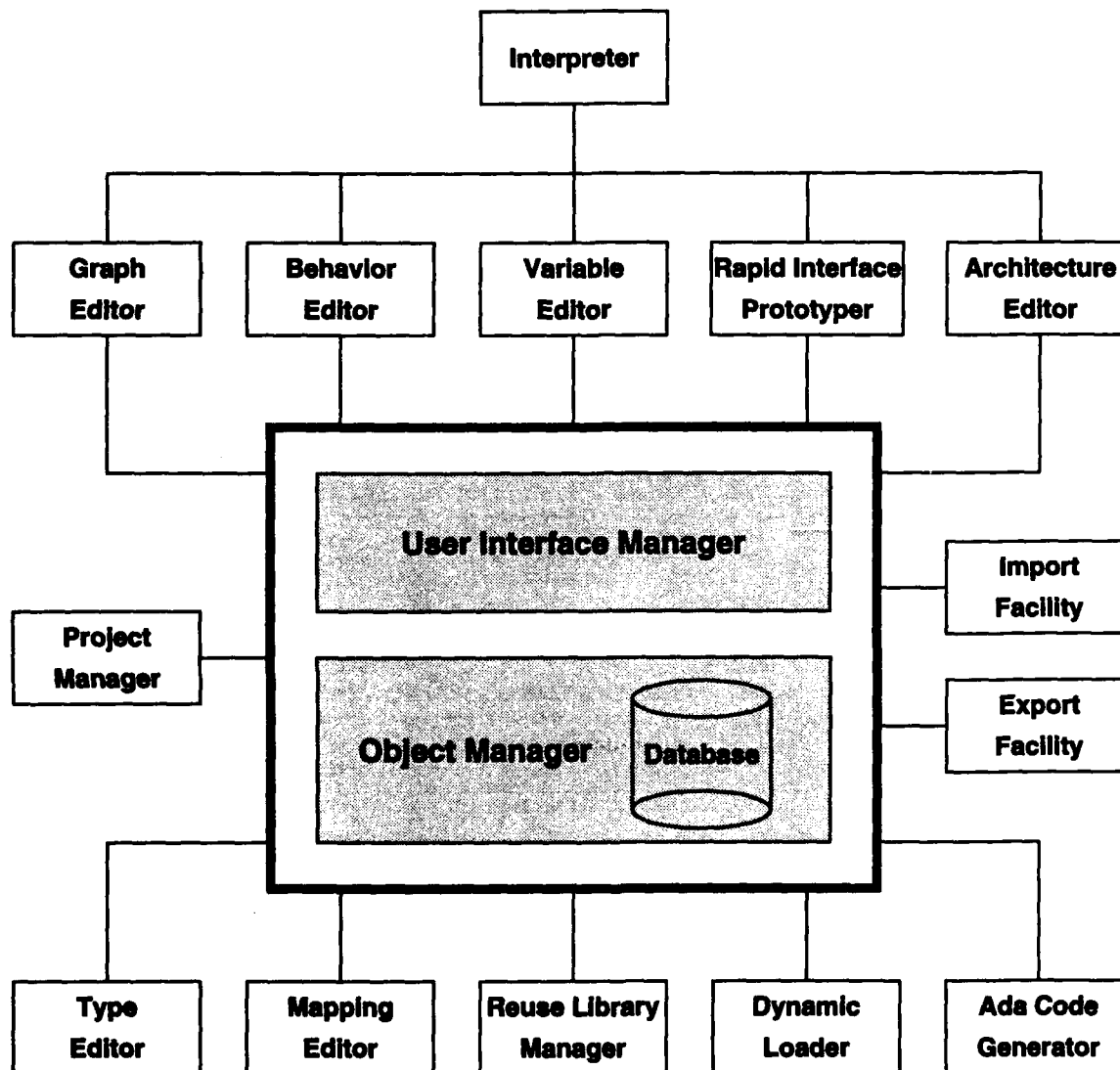


Figure 5. Proto High-Level Block Diagram

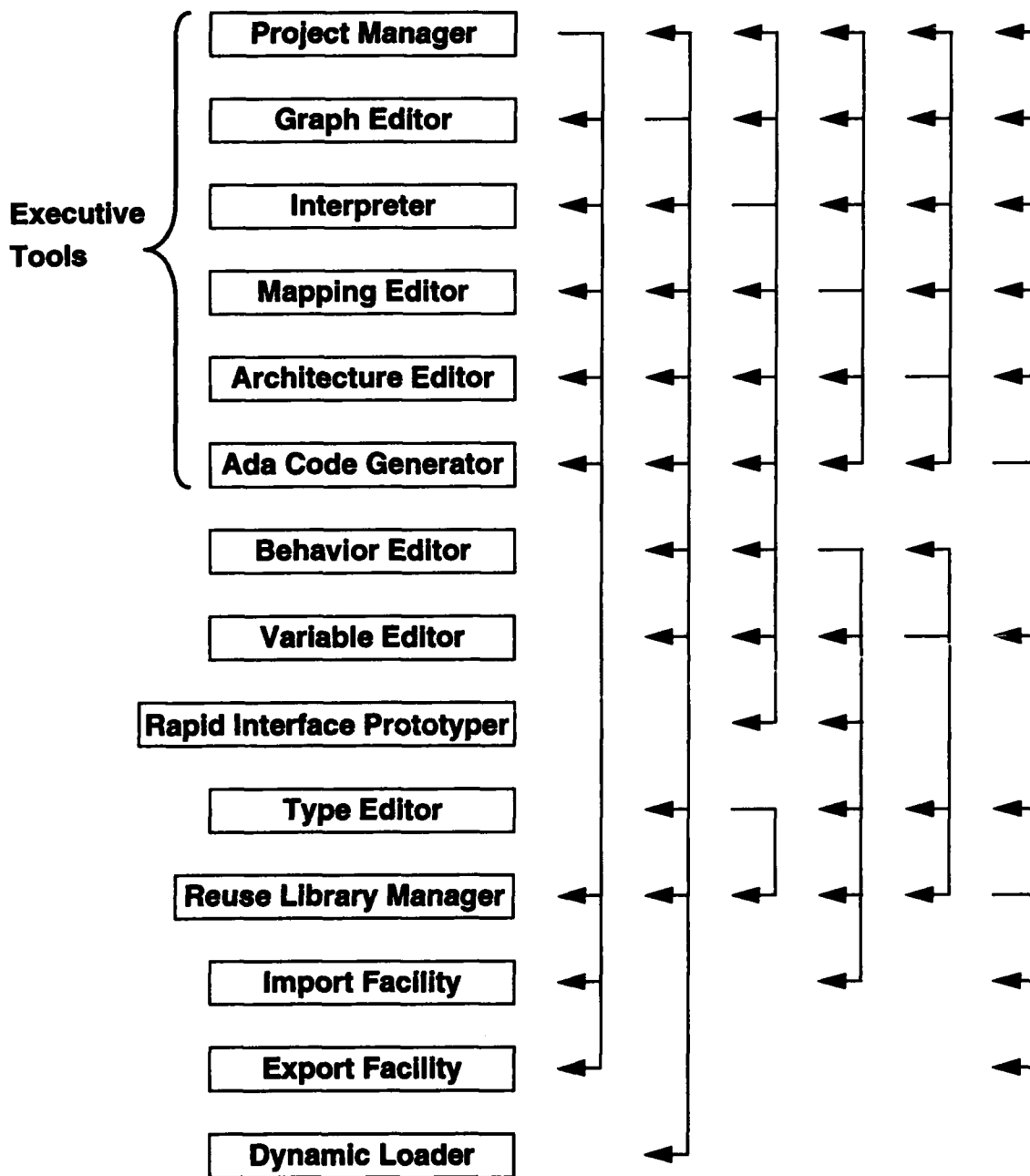


Figure 6. Proto Primary Tool Access Paths

4.2.3 Behavior Editor

Behaviors for SSDL nodes are constructed using a full-function text editor that is accessible from the Graph Editor. This editor implements a simple interface for inserting syntax templates of SSDL language constructs to assist users in developing programs. It is possible for multiple Behavior Editors to be opened simultaneously since a different Behavior Editor instance is created for each SSDL node. A parser for SSDL behaviors, which generates an intermediate SSDL object representation, is integrated with the Behavior Editor. As part of supporting a variety of conventional mouse-directed text editing commands, the Behavior Editor allows writing/reading behavior text to/from files under user control. In addition, scrolling lists are used to provide access to built-in mathematical functions, dynamically loaded user functions, and functions that are used to manipulate user interface prototypes constructed with RIP. The Behavior Editor can also be opened from the Interpreter in read-only mode.

4.2.4 Type Editor

The Type Editor is a multiwindow tool for creating and modifying library components, including enumeration, class, and array types, and library process nodes. (Recall that a library that is part of a project is called a project library.) The editor uses a browserlike approach to display information for multiple types simultaneously. Integrated access to this editor can be obtained from the Graph, Variable, and Behavior Editors, and the Reuse Library Manager. Data types are library-specific; that is, they are not automatically shared between libraries. It is possible, however, to use the Export and Import Facilities to merge components from one library with those of another.

For enumerations, operations supported include adding enumerators, deleting enumerators, and specification of a default enumerator for an enumeration type. For arrays, features that can be specified by a user include number of dimensions, upper/lower bounds, and element type. In addition, array cross sections can be specified for setting default array element values. For classes, type features that can be specified are attribute names, attribute types, attribute default values, methods, and inheritance.

The Type Editor for classes also provides a graphical display of class hierarchies to help the user visualize inheritance relationships. Operations supported by this display include automatic tree layout, panning, zooming, scrolling, and vertical or horizontal orientation. A method viewer that lets users create and edit methods is also an integral part of the Type Editor for classes.

The Type Editor implements viewing and editing of aggregate values (i.e., arrays and classes) by providing inspector windows that can be opened to any depth of a complex data structure. The Type Editor provides options for configuring the scrolling list ordering, the scrolling list sizes, and the information to be displayed (e.g., attribute types). Fast-path options are also available for simplifying repetitious editing tasks (e.g., changing attribute default values).

Another capability provided by the Type Editor is an editor for defining library process nodes. This editor is similar in functionality to the method viewer, except that library process nodes are not associated with any specific class type.

4.2.5 Variable Editor

A multiwindow Variable Editor is used for creating and/or editing SSDL variables, which include ports, locals, and stores. This editor uses a browserlike approach to display information for

multiple variables simultaneously. The Variable Editor is accessed directly from the Graph and Behavior Editors, from the Interpreter (for viewing current variable values during simulation), and from the Reuse Library Manager. Direct access to both the Behavior and Type Editors is available from the Variable Editor, which simplifies accessing information associated with the current editing context. For example, when the Variable Editor is being used to edit the locals of a particular leaf node, invocation of the Behavior Editor provides immediate access to that node's behavior without having to use the Graph Editor interface.

The Variable Editor provides a number of user-specified options for configuring the contents of its windows. Configurable items include scrolling list ordering, scrolling list sizes, and information to be displayed (e.g., variable types, store locks, port directions, initial value, current value). Fast-path options are also available for simplifying repetitious editing tasks (e.g., setting initial array element values). As with the Type Editor, array cross sections can be specified for setting initial values for multiple array elements simultaneously. The Variable Editor also provides functionality for easy navigation through multiple layers of composite data values.

Another capability provided by the Variable Editor is an editor for editing control ports. This editor is primarily used for associating Boolean expressions with control output ports, since control ports do not have data types or values. These expressions are evaluated during interpretation when the end of a behavior is reached and, if TRUE, cause a control signal to be automatically generated on the control output port. The control port editor lets a user define Boolean expressions for control output ports and contains a parser to validate such expressions.

4.2.6 Dynamic Loader

Proto contains a Dynamic Loader tool that lets a user dynamically load and link C functions that have been developed and compiled externally. These functions can be subsequently invoked from SSDL process node behaviors and provide a convenient means for reusing programs that have been developed previously. The Dynamic Loader, which is invoked from the Graph Editor, allows specification of function names, parameter types, and object files where the user functions are defined. Object files are obtained by compiling C source files.

4.2.7 Rapid Interface Prototyping System

Proto's user interface editor function is implemented by providing integrated access to the REED RIP tool for construction of graphical application scenarios. This tool supports building input and output interfaces to SSDL prototypes. RIP supports not only interface graphics but also forms-based devices, such as buttons and text fields. In this manner, generic and application-specific user-interface prototypes can be developed in conjunction with functional prototypes.

A functional API to manipulate RIP objects by referencing and modifying their state attributes is provided for SSDL behaviors. Proto uses this interface to invoke and connect to a RIP interface, modify the state of user interface objects, get input values from an interface, and wait for user interface actions. To improve performance and provide better synchronization, communication between RIP and Proto can take place over both buffered and unbuffered connections.

The reader is referred to REED documentation for details of the RIP tool functionality [ISSI 1993].

4.2.8 Reuse Library Manager

The Reuse Library Manager is invoked directly from the Graph Editor. Each Proto database contains multiple component libraries. Some of these libraries, referred to as project libraries, are contained inside projects. SSDL constructs that can be stored in domain-specific reuse libraries include both leaf and composite process nodes, class types (including methods), array types, and enumeration types. Both stand-alone and project libraries are edited using the Type Editor. Other operations that can be used to create and modify libraries include importing and merging of libraries. These operations permit easy access to sharing of reusable component libraries between domain users, systems analysts, and domain experts. Merge involves copying and integrating all types, method nodes, and process nodes contained in a library with those contained in another library.

Graphical copy-and-paste and cut-and-paste operations are the primary means for inserting design components from the reuse library into a design. For method and process node cut and copy operations, a design component is defined as a process node plus its behaviors, ports, and refinements (including internal nodes, behaviors, ports, stores, and connections). SSDL facilities for reuse component customization include explicit control sequencing mechanisms (including control expressions) and the inherent flexibility of class methods to be called with subclass and superclass instances of a class hierarchy.

The Reuse Library Manager also provides a search capability to query and obtain listings of subsets of library components via keyword search. Components that can be selectively searched include classes, attributes, method nodes, arrays, enumerations, and process nodes. Additionally, keyword searches can be conducted on node description strings that are constructed using the Graph Editor. Options for keyword search include case sensitivity, substring search, and searching for n out of m keywords.

4.2.9 Export Facility

The Export Facility is Proto's tool for generating ASCII file representations of SSDL projects and libraries. This file representation can handle all dataflow, behavior, and data specification aspects of an SSDL design, including graphical layout information. The Export Facility can be accessed from either the Project Manager or the Reuse Library Manager.

4.2.10 Import Facility

The Import Facility tool recreates a project or library from an exported ASCII file representation of the project or library. The tool can recreate all dataflow, behavior, and data specification aspects of an SSDL design, including graphical layout information. Since the export/import format is public, it can be used to import design representations from tools other than Proto.

4.2.11 Architecture Editor

Proto users edit architecture definitions using an object-oriented, window-based graphical Architecture Editor. Hardware primitives that can be created and manipulated include processors, memories, and buses. As in the dataflow Graph Editor, editing operations in the Architecture Editor consist of selecting an object and selecting the action to be performed on that object. These actions include inserting, moving, deleting, labeling, and connecting objects. Other graphical facilities available in the editor include panning, zooming, and scrolling.

The processor, memory, and buses can be configured in a variety of ways to allow modeling of many kinds of MIMD machines. Architectures that can be defined include shared-memory multiprocessors, distributed-memory multiprocessors, and hybrid machines consisting of shared-memory and distributed-memory components. A facility for automatically laying out parameterized mesh architectures, consisting of rows and columns of processors interconnected by buses, is also provided.

In addition to network topology, users can specify parameters for architecture resources using menus. These parameters include processor execution times for SSDL operations, memory-access times, and bus delays specified in terms of simulation time units. It is also possible to associate resource utilization instruments (e.g., digital display and thermometer) with each architecture component to monitor dynamic resource usage during simulation. Minimum bus-delay static routing between architecture components is computed using Floyd's shortest path algorithm [Aho et al. 1983].

4.2.12 Mapping Editor

Architectural resource models are incorporated into software models by establishing mappings from software (logical) components to hardware (physical) components. Mappings describe a software-to-hardware allocation and consist of (process node, processor) and (store, memory) pairs. Thus, mappings are embeddings of dataflow graphs into resource graphs. The Mapping Editor is used to create, rename, and delete mappings. More than one mapping can be specified for a given dataflow graph; each mapping can be associated with one or more different architecture definitions. That is, you can use the same named mapping with different architecture definitions, although at a given point in time only one such architecture may be selected.

Mapping pairs are added to a software-to-hardware allocation using interactive menus accessible from the Graph Editor. The Mapping Editor automatically performs consistency and completeness checks to detect illegal mappings. For example, attempts to map a node to a memory are flagged as illegal. In addition, the Mapping Editor supports heuristic and random algorithms for automatic allocation of software components to hardware components. These algorithms are automatically called during simulator initialization for mapping all unallocated software components.

Typically, mappings in Proto are defined in the context of an explicitly specified architecture definition that is set in the Mapping Editor. For early stages of functional prototype development, the Mapping Editor also supports specification of default sequential and fully parallel mappings. In the *sequential mapping*, all nodes are mapped to a single processor (i.e., node executions are serialized), all stores are mapped to a single memory that has read and write access times of zero simulation time units, and the processor is connected to the memory by a zero-delay bus. In the *fully parallel mapping*, each node is mapped to its own processor, all stores are mapped to a single memory that has read and write access times of zero simulation time units, and the processors are connected to the memory by a zero-delay bus. The fully parallel mapping models true dataflow execution.

4.2.13 Interpreter

Prototype execution is implemented by a simulation-based Interpreter consisting of a simulation kernel, behavior interpreter, architecture modeler, and scheduler. The simulator uses the Graph,

Variable, Behavior, RIP, and Architecture Editor interfaces, although prototype editing is disabled during its execution.

The simulation kernel implements the simulation cycle, which is the backbone that references other components of the simulator. This cycle consists of update and execution phases. In the update phase, event processing generates updates of stores, port queues, and instrumentation displays; interpretation of behavior triggers; and scheduling of node behaviors. Scheduled behaviors are interpreted in the execution phase. An efficient event queue structure manages all time-ordered events generated by the system. Resource modeling is incorporated into the simulation kernel in a manner that minimizes the impact on interpreter and scheduler functionality. This is facilitated by the modularity of the simulation cycle and single event queue design.

The SSDL behavior interpreter executes all SSDL constructs, implements access to RIP through the API integration functions, provides graphical output and input/output forms for viewing and entering user data, and keeps track of execution time. The interpreter also generates events for message-passing and store accesses. Another function of the interpreter is to dynamically collect execution statistics, such as the number of times a node is executed.

A sequential scheduling algorithm, based on priorities, activation times, and execution counts of dataflow graph nodes, arbitrates among multiple nodes competing to execute on a single processor. Node priorities are assigned to the dataflow graph nodes using the Graph Editor. Since software-to-hardware mappings are specified statically, execution-time allocation is unnecessary. The scheduler also manages the scheduling of multiple parallel processors, although this does not involve sophisticated functionality because software/hardware allocation is specified statically.

The simulator includes an architecture modeler module that performs the tasks of *memory* and bus serialization, message and memory-access routing, and hardware statistics collection. Note that while memory and bus requests are serialized by the architecture modeler, the scheduler is responsible for serializing node requests for processor resources. Using the static routings computed by the Architecture Editor, the modeler generates appropriate events to process message-passing and store access (read/write) requests generated by the interpreter. Thus, logical message-passing and store access operations are implemented as high-level constructs. Consequently, a software design does not have to be changed when it is ported from one architecture to another, or when different mappings are used for the same architecture. For example, a software specification that assumes a logical shared-memory model can be mapped onto a shared-memory multiprocessor or a distributed-memory multiprocessor without changing any code. In the latter case, the physical model is one of *distributed shared memory*.

Graphical instrumentation of a prototype simulation is an important function of the simulator. Dynamic mechanisms for assisting users in observing prototype execution include dataflow graph animation via object highlighting, textual data displays, special-purpose graphical displays (e.g., queue status icons), and general-purpose displays (e.g., thermometers and cartesian plots). Instrumentation for dynamically displaying resource utilization statistics using thermometers and digital displays is also available.

In addition to instrumentation, several other debugging mechanisms are available. For example, the interpreter implements setting breakpoints before and after node execution; single-stepping of node behaviors; execution of the SSDL display, input, and input_output statements (for variable values); and interrupting the simulation. Furthermore, most objects in the system can be browsed

when prototype simulation is suspended. Also, the Behavior, Variable, and Type Editors can be accessed for examining prototypes, although editing is disabled while in the Interpreter tool.

Additional debugging tools that are useful for identifying problems with parallel and distributed system prototypes are targeted towards using event trace information for detecting deadlocks and other exceptional event sequences (e.g., data unavailability, data corruption) using conditional breakpoints. For performance evaluation, comprehensive dataflow graph and resource model performance statistics can be generated whenever simulation is suspended and are automatically generated when simulation completes.

4.2.14 Ada Code Generator

An integrated Ada Code Generator tool can be invoked to generate an executable Ada program for a given SSDL project. All SSDL constructs are supported (including access to external C functions), except the Unspecified data type. The Ada Code Generator provides options for inserting debug statements to print a trace of the generated code execution sequence.

The generated code uses multiple tasks to implement SSDL processor concurrency, providing a model suitable for distributed systems. Ada's rendezvous mechanism is used for communication and synchronization between tasks. To guarantee consistency between results obtained during SSDL prototype simulation and generated code execution, the scheduling algorithm used for the generated code is the same as that employed by the interpreter. However, consistency between models that depend on execution times and inherent nondeterminism found in SSDL behaviors is not guaranteed.

Concurrency in the generated Ada is based upon the currently selected mapping for an SSDL prototype. Each architecture processor is implemented as a package that contains executive tasks, input buffer tasks, output buffer tasks, and a scheduling procedure. Each node is also implemented as a package that contains variables to implement the ports and locals associated with the node, and procedures to implement behavior actions. Processor packages reference the appropriate process node packages, corresponding to SSDL node allocation, by "withing" them. Generated code execution is initiated by a main procedure that calls initialization procedures for each processor package, which in turn results in calls to initialization procedures for the node packages. The main procedure then starts each processor's executive task loop by issuing entry calls that allow execution to proceed.

5 METHODOLOGY OVERVIEW

Proto's tool suite allows rapid system prototyping of functional specifications, user interfaces, and hardware architectures. These facilities support conducting functional and performance feasibility studies during the requirements specification stage of the software development cycle. Proto also contains functionality for automatic generation of code from functional specifications. Combined with an iterative prototyping methodology, the Proto environment provides significant benefits in reducing requirements definition errors of proposed concurrent systems, which subsequently reduces the cost and risk involved in developing such systems.

The Proto methodology (or method), described in [Acosta and Beal 1994], is intended to provide recommended procedures and guidelines for using Proto to support the requirements specification stage of the software lifecycle.

- *The methodology's primary emphasis is on using the facilities provided within Proto to support rapid prototyping and validation of requirements.*

Modeling activities covered by the methodology are performed to gain a greater understanding of complex software systems and lessen the impact that ambiguous, inconsistent, and incomplete requirements have on subsequent stages of the system development process.

Given the focus on rapid element prototyping embodied in Proto, the methodology is obviously directly suitable for use in a spiral development process model. Prototypes developed following the methodology are, for the most part, intended to be "throwaway" products built solely for the purpose of reducing design and development risk by supporting the requirements engineering function. Nonetheless, since the spiral model accommodates most other process models, the methodology is easily customized to many other development models. One should thus view the methodology as describing the "how" for the prototyping portion of the requirements specification stage of some software process model, whereas the "what" and "when" is defined in the model itself.

The methodology is an iterative procedure that includes activities associated with requirements review, prototype problem selection, prototype construction, prototype evaluation, and prototype revision. These activities are incorporated into 9 *steps* illustrated by Figure 7. These steps that are briefly characterized below:

1. **Requirements Review** - Collection and review of relevant domain and problem material, including requirements specification (if any). Interviews with users. Definition of problem and system goals.
2. **Prototype Problem Selection** - Analysis of requirements to determine areas of risk. Selection of components to prototype for reduction or elimination of risks. Strategy, plan, and test scenarios for prototyping activities. Determination of when the prototyping methodology should be exited.
3. **Functional Prototype Construction** - Object-oriented analysis through the identification of classes, responsibilities, and collaborations. Construction of an executable SSDL prototype in Proto using object-oriented techniques and reuse. Identification of a top-level entry point to initiate prototype execution.
4. **User Interface Prototype Construction** - Construction of a user interface prototype using RIP, including overlay structure, graphic objects, bitmap images, widgets, and maps. Designation of names and switch actions used to interface with the Proto functional prototype.
5. **Resource Prototype Construction** - Construction or reuse of one or more hardware architecture resource models. Parameterization and scaling of components of the resource models.
6. **Prototype Integration** - Specification of classes in the functional prototype to integrate with objects in the user interface prototype. Definition of software/hardware mappings that allocate software components in the functional prototype to architecture components in the resource prototype.

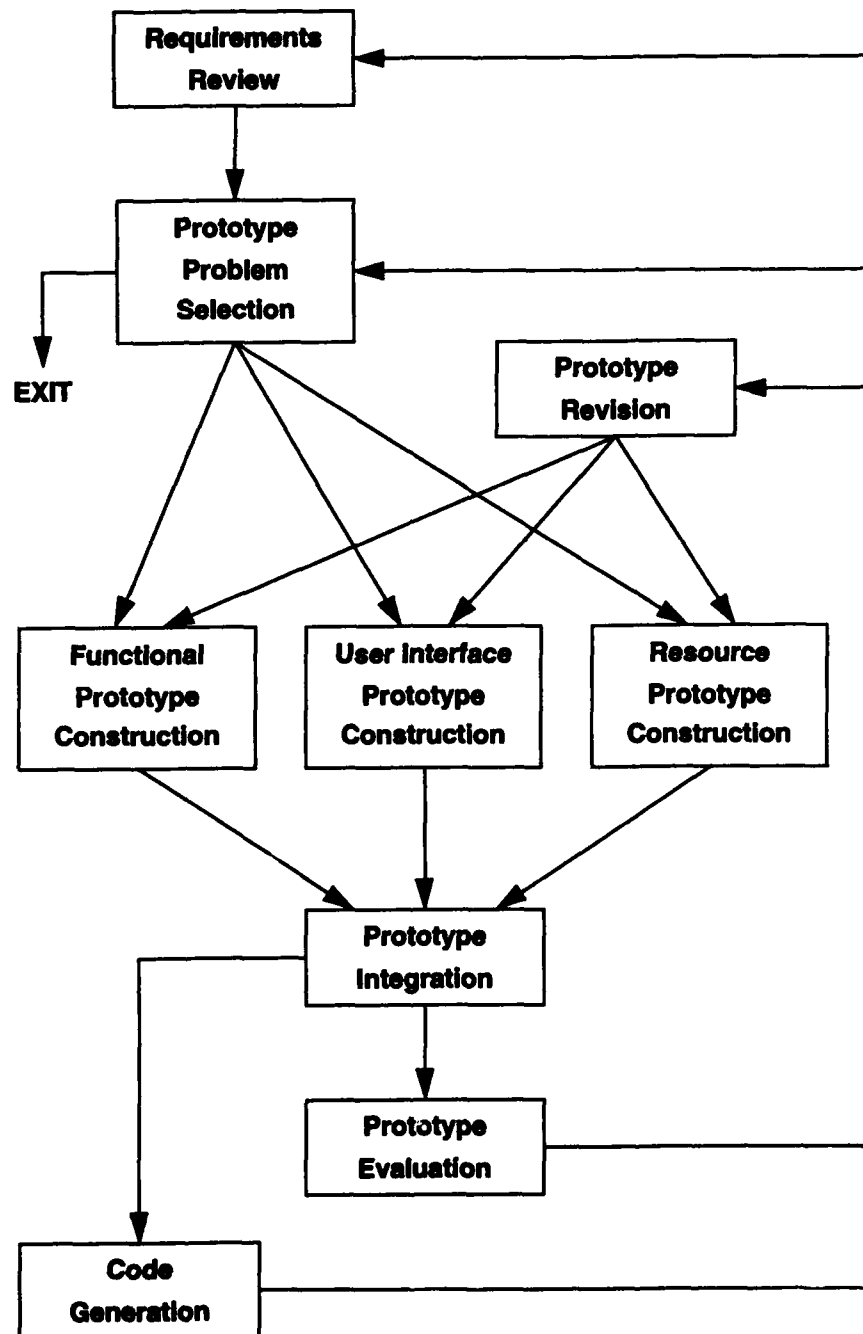


Figure 7. Proto Methodology Steps

7. **Prototype Evaluation** - Exercising of the functional, user interface, and resource prototypes for debugging and end user analysis. Identification of features to be included in future prototype revisions. Identification of changes that need to be made to the system requirements. Determination of how to proceed with the prototyping activities.
8. **Prototype Revision** - Updating of component features to be prototyped. Revision of strategy and plan for prototyping procedure.
9. **Code Generation** - Generation and compilation of code from the SSDL prototype. Exercising of the prototype, possibly on target platform, using more extensive test cases. Integration of prototype with other operational subsystems to validate requirements in a larger scenario.

A significant feature of the methodology is that it adopts an object-oriented analysis approach to prototype development within Proto. As implemented in Proto, SSDL is really a hybrid dataflow and object-oriented prototyping language. We have taken the object-oriented analysis tack because experiences with Proto have shown that use of object-oriented techniques lead to prototypes that are easier to understand, modify, and reuse. And, even though the prototypes will be eventually thrown away, object-oriented prototypes will reduce the complexity of rapid changes and modifications that may be required in satisfying user requests as part of the prototyping activities. Plus, analysts may want to employ reuse techniques to speed up the prototyping process by saving prototype elements in a reuse library for future retrieval. The next chapter of this document provides additional background on the use of object-oriented techniques, including definitions of all relevant terms.

The object-oriented analysis activities contained in the Proto methodology are derived from the CRC technique described in the book *Designing Object-Oriented Software* by Wirfs-Brock, Wilkerson, and Wiener [Wirfs-Brock et al. 1990]. CRC stands for classes, responsibilities, and collaborations. These represent the 3 primary tasks involved in object-oriented development:

1. **Classes** - Find all the objects that comprise the problem.
2. **Responsibilities** - Determine the intrinsic properties of each object (i.e., activities an object performs and data it maintains).
3. **Collaborations** - Determine the extrinsic properties of each object (i.e., interdependencies between objects in performing responsibilities).

Tasks 1-3 are usually repeated several times as understanding of the problem and refinements become available. In most applications, an additional one-time task is performed:

4. **Application** - Identify the top-level entry point (i.e., collaboration) for the application. (This step is not explicitly stated in CRC.)

The CRC technique is well suited to Proto. Proto did not start out as an object-oriented system but rather is based on dataflow techniques that had evolved to support functional decomposition. It is only with the latest releases of Proto that object-oriented features have been added, as well as extending support for further dataflow and control flow features (e.g., persistent stores, control connections). The added features can help considerably in structuring and maintaining a prototype application. Despite system support for object-oriented language features in SSDL, however, there is no special support for analysis and design features. Many of the newer methodologies [Rumbaugh et al. 1991; Booch 1991; Graham 1991] utilize specialized notations unsupported by

the current set of Proto tools and concepts unsupported by the current implementation. While such notations do help in documenting the relationships between objects, they do not provide much assistance in creating objects in the first place: coming up with the classes, determining their functionality, and how they interact with each other.

The variation on CRC adopted by the Proto methodology requires little more than the various features of the data editors for construction of the object model. When combined with Proto's reuse capabilities, it provides the means for building prototypes quickly and assuredly while using good development techniques.

Although CRC lacks some of the "flash" of later methodologies, it has the virtues of simplicity, understandability, and flexibility, particularly to newcomers to object-oriented methods. The use of CRC in the Proto methodology is sufficiently generic and compartmentalized that other object-oriented methods could be tailored for application within the Proto methodology with little or no effect on steps other than that of functional prototype construction. Moreover, because of Proto's roots in dataflow specification and design, the methodology could really be considered a hybrid of object-oriented and functional decomposition approaches.

6 EXAMPLE

Several of the functional and performance prototyping capabilities of Proto are now illustrated using a simple example involving the architecture and design of a distributed electronic funds transfer (EFT) system. The EFT specification was developed from a set of requirements adapted from the description of the system in [Staskauskas 1988]. A dataflow diagram of the system, developed using PProto, appears in Figure 8.

The EFT system processes *transactions* involving the automatic transfer of funds from *customer accounts* to *merchant accounts*. The following three kinds of nodes are used in the system:

- **Terminal.** Originates new transactions and notifies customer whether transactions are accepted or rejected.
- **Bank.** Debits customer accounts and credits merchant accounts according to transaction requests.
- **Switch.** Routes transactions between terminals and banks according to customer and merchant account numbers and transaction status.

A transaction originates at a point-of-sale *terminal* and contains information regarding the transfer amount, customer account number, and merchant account number. When a transaction arrives at the *switch* from a terminal, it is routed to the appropriate customer *bank*. Subsequently, the bank then either accepts the transaction and debits the customer account with the amount of the transaction, or rejects the transaction if there are insufficient funds in the customer account. The bank then forwards the transaction to the switch.

If the transaction is rejected, the switch sends the transaction back to the originating terminal. If the customer bank accepts the transaction, it is routed to the merchant account bank. Upon arrival at this bank, the merchant account is credited with the transaction amount. The transaction is then sent to the switch for routing back to the originating terminal.

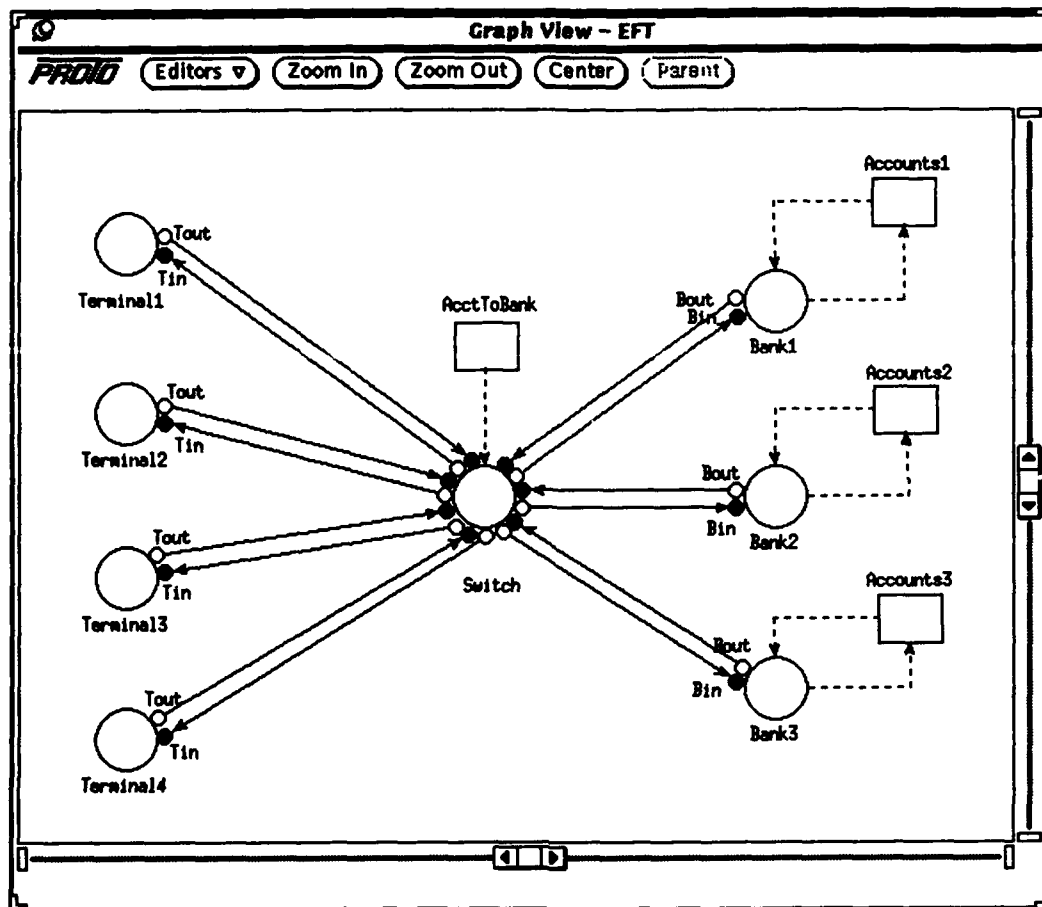


Figure 8. EFT Dataflow Graph

In addition to the three kinds of nodes mentioned above, there are two types of data stores required for the EFT system:

- **AcctToBank.** Array used by the switch to map customer and merchant account numbers to banks.
- **Accounts.** List of account numbers and balances contained at a bank.

This example is simple enough that node refinements are not necessary to capture the initial specification being considered. Although not illustrated here, future iterations of the design cycle would use functional and performance simulation results to refine and decompose some of the functions (e.g., switch routing) and data structures of the system.

The design of Figure 8 constitutes an executable SSDL prototype of the EFT system. The Proto simulator is used to verify the prototype's functionality. In addition to the simulation functions implemented by the SSDL interpreter and scheduler, the debugging, instrumentation, and animation facilities of the system can be used to observe architectural characteristics of the system

(e.g., message communication, data store accesses) and verify functionality. Transaction data enters the system from the terminals by calling C functions that read data files. It is thus possible to easily exercise the system for different data sets.

Particularly useful is the ability to automatically assign each node to a different processor using a fully-parallel mapping. The model obtained by this approach is an ideal distributed EFT system in which communication and memory access costs are eliminated. Not only does this allow verifying the functional capabilities of the system, but simulation results can be used to suggest target architectures and software/hardware mappings that are best matched to the EFT architecture. Also, such prototyping helps identify serial bottlenecks that can be reduced through further functional changes.

Performance simulation continues with mapping of the EFT software specification to one or more hardware architecture definitions. Several mappings and architectures are suggested in Figures 10-12. Figure 13 contains simulation time measurements for these architectures using the same transaction input data. Measurements for the default sequential and fully-parallel architectures are also included. The same processor model is employed for each architecture. Results of three simulation runs are shown for the shared-memory, distributed-memory, and hybrid machines using bus and memory access delays of 0, 1, and 10 simulation time units.

By analyzing simulation time, resource utilization, and other performance characteristics of the combined software/hardware models, it is possible to identify advantages and disadvantages of different system architectures in early phases of the specification and design cycle. For example,

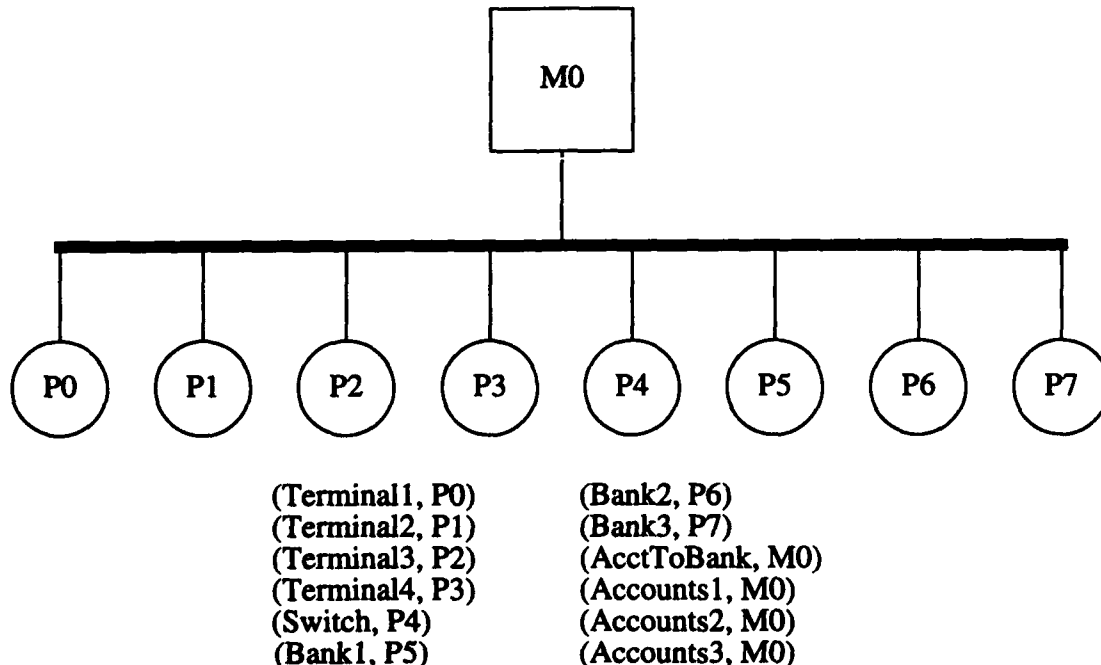


Figure 9. EFT Mapping to Shared Memory Machine

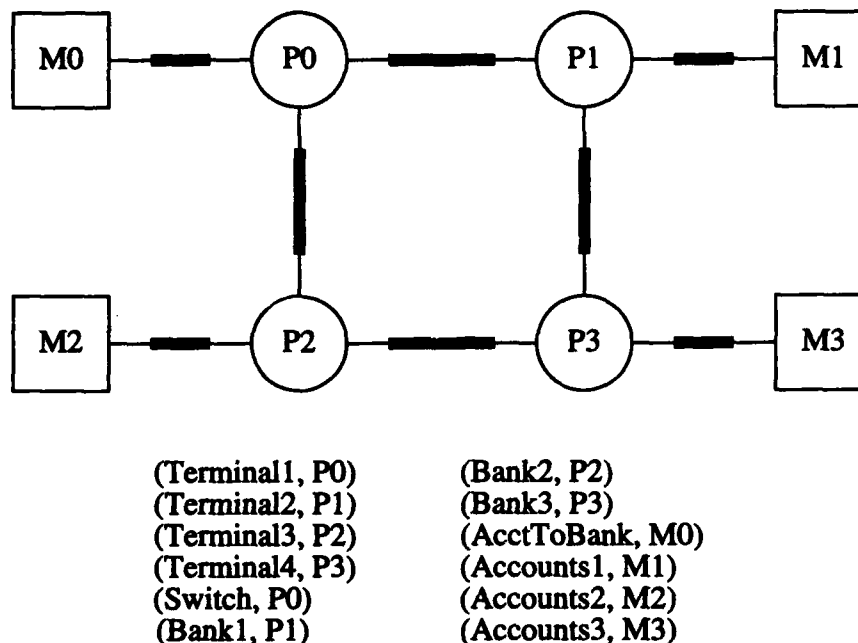


Figure 10. EFT Mapping to Distributed Memory Machine

with bus and memory access times of 0 units, simulation time for the fully-parallel architecture is the same as that of the shared-memory and hybrid machines. The distributed-memory machine takes longer due to the allocation of two nodes to each processor. Simulation times for the three machines are relatively close when bus and memory access times of 1 unit are employed – bus utilization is negligible in all three machines.

Once access times are increased to 10 time units, bus utilization for the shared-memory machine jumps to almost 90% which accounts for the additional simulation time used by this machine. On the other hand, even though the distributed-memory machine has only half of the processors available in the hybrid machine, its performance is competitive due to greater bus connectivity and balanced node distribution.

7 CONCLUSIONS

Proto is a simulation-based rapid prototyping system for conducting functional, user interface, and performance feasibility studies during the requirements specification stage of the software development lifecycle. Incorporation of rapid prototyping techniques as part of the software development lifecycle helps eliminate ambiguities, inconsistencies, and incompleteness in requirements. Increased preciseness of requirements, in turn, leads to improved quality in delivered software products, as well as reduced cost and improved predictability of schedules.

Proto supports codesign and analysis of high-level software and hardware application architectures. Proto implements the SSDL high-level specification prototyping language. SSDL

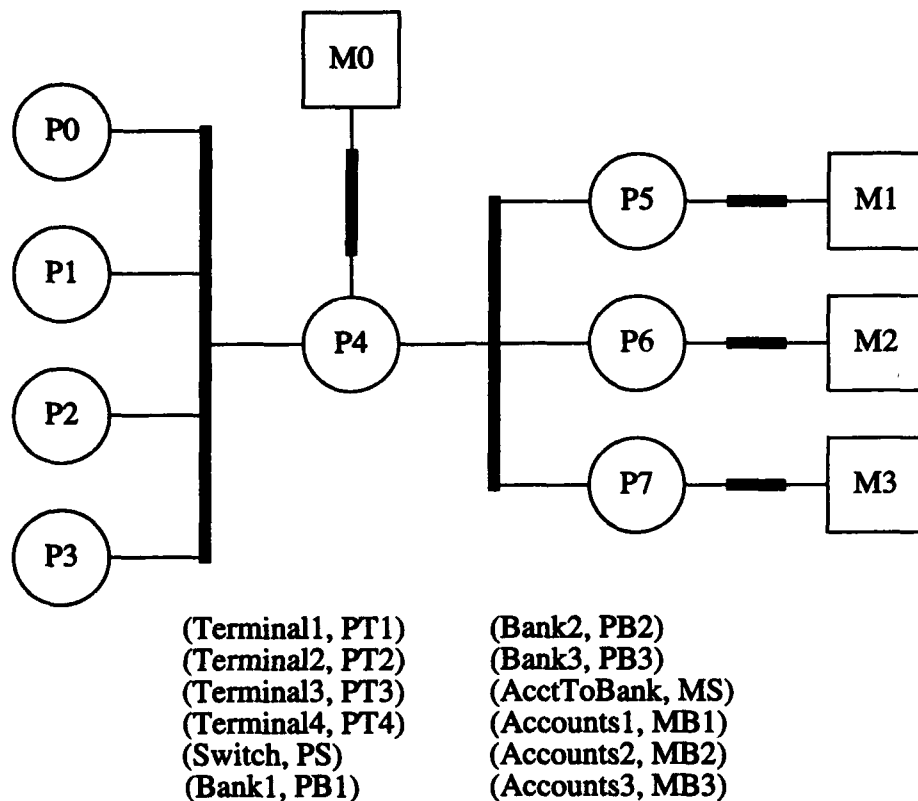


Figure 11. EFT Mapping to Distributed Hybrid Machine

primitives are flexible enough to model a variety of concurrent, parallel, and distributed systems, including those based on message-passing and shared-memory synchronization and communication mechanisms. The Proto system contains a variety of prototyping tools including a graphical dataflow editor, type and variable editors, hardware resource modeler, dynamic loader, reuse facility, user interface editor (based on accessing RIP), interactive simulator, and Ada code generator.

8 REFERENCES

- Acosta, R. D. 1991. *Final report for Parallel Proto*. Technical Report No. ISSI-A89A00006. Austin, TX: International Software Systems, Inc., June. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-89-C-0129, CDRL A013.
- Acosta, R. D. 1992. Specification prototyping of concurrent Ada programs in DProto. *Proceedings of TRI-Ada '92* (November), 258-266.
- Acosta, R. D. 1993a. *Functional description for Proto*. Technical Report No. ISSI-C91A00004C. Austin, TX: International Software Systems, Inc., December. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A004.

Architecture	Bus/Memory Access Times		
	0 time units	1 time unit	10 time units
Shared-Memory	3620	4108	10142
Distributed-Memory	4153	4566	8828
Hybrid	3620	4077	8361
Default Sequential	8746	—	—
Default Fully-Parallel	3620	—	—

Figure 12. EFT Example Simulation Results

- Acosta, R. D. 1993b. *System/subsystem specification for Proto*. Technical Report No. ISSI-C91A00005B. Austin, TX: International Software Systems, Inc., December. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A005.
- Acosta, R. D., and R. J. Beal. 1994. *Proto Methodology*. Technical Report No. ISSI-C91A00012A. Austin, TX: International Software Systems, Inc., January. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A012.
- Acosta, R. D., and G. Liu. 1993. *Demonstration Plan for Proto*. Technical Report No. ISSI-C91A00010A. Austin, TX: International Software Systems, Inc., December. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A010.
- Acosta, R. D., C. L. Burns, W. E. Rzepka, and J. L. Sidoran. 1994. A case study of applying rapid prototyping techniques in the Requirements Engineering Environment. To appear in *Proceedings of the 1994 International Conference on Requirements Engineering* (April).
- Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1983. *Data structures and algorithms*, 208-212. Reading, MA: Addison-Wesley.
- Boehm, B. W. 1988. A spiral model of software development and enhancement. *Computer* 21 (5): 61-72.
- Booch, G. 1991. *Object Oriented Design*. Redwood City, CA: Benjamin/Cummings.
- Box, B. D. 1993. *DProto demonstration plan and reuse library description*. Technical Report No. TM(H)030/001/00. Huntsville, AL: Optimization Technology, Inc., December.

- Burns, C. L. 1991. Parallel Proto - A prototyping tool for analyzing & validating sequential and parallel processing software requirements. *Proceedings of the IEEE Second International Workshop on Rapid System Prototyping* (June), 151-160.
- Cechovic, S. C. 1993. *Final DProto reuse process report*. Technical Report No. TM(H)019/003/00. Huntsville, AL: Optimization Technology, Inc., April.
- Gordon, V. S., and J. M. Bieman. 1991. *Rapid prototyping and software quality: Lessons from industry*. Technical Report No. CS-91-113. Fort Collins, CO: Colorado State University Department of Computer Science, July.
- Graham, I. 1991. *Object Oriented Methods*. Reading, MA: Addison-Wesley.
- Hartman, D., M. Konrad, and T. Welch. 1988. *VHLL system prototyping tool final report*. Austin, TX: International Software Systems, Inc. Delivered to Rome Air Development Center, Griffiss Air Force Base, NY, under contract F30602-85-C-0124.
- ISSI. 1992. *Final technical report for Proto+*. Technical Report No. ISSI-C88A00011. Austin, TX: International Software Systems, Inc., March. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-88-C-0029.
- ISSI. 1993a. *Software user's manual for the Rapid Interface Prototyping system (RIP) of the Requirements Engineering Environment*. Technical Report No. ISSI-B89A00011A, Volume 2. Austin, TX: International Software Systems, Inc., January. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-89-C-0200, CDRL A011.
- ISSI. 1993b. *Software test plan for Proto*. Technical Report No. ISSI-C91A00008B. Austin, TX: International Software Systems, Inc., November. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A008.
- ISSI. 1993c. *Test analysis report for Proto*. Technical Report No. ISSI-C91A00009. Austin, TX: International Software Systems, Inc., December. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A009.
- ISSI. 1993d. *Proto user's manual*. Air Force Contract No. F30602-91-C-0012, Technical Report No. ISSI-C91A00014F. Austin, TX: International Software Systems, Inc., November. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-89-C-0129, CDRL A012.
- ISSI. 1994. *Program specification for Proto*. Technical Report No. ISSI-C91A00006. Austin, TX: International Software Systems, Inc., February. Delivered to Rome Laboratory, Griffiss Air Force Base, NY, under contract F30602-91-C-0012, CDRL A006.
- RADC. 1990. *Statement of work for Proto code generation technique*. Air Force Contract No. F30602-91-C-0012. Griffiss AFB, NY: Rome Air Development Center, October.
- Rzepka, W. E., and Y. Ohno. 1985. Requirements engineering environments: Software tools for modeling user needs. *Computer* 18 (4): 9-12.
- Rzepka, W. E. 1992. A requirements engineering testbed: Concept and status. *Proceedings of the Second International Conference on Systems Integration* (June), 118-126.

- Rzepka, W. E., J. L. Sidoran, and D. A. White. 1993. Requirements engineering technologies at Rome Laboratory. *Proceedings of the IEEE Symposium on Requirements Engineering* (January), 15-18.
- Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall.
- Sidoran, J. L., and R. D. Acosta. 1993. Use of simulation techniques in a prototyping environment for requirements engineering. *Proceedings of the 1993 International Simulation Technology Conference* (November).
- Wirfs-Brock, R., B. Wilkerson, and L. Wiener, L. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.

***MISSION
OF
ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.